



## Hoofdstuk 17

### Java SE 8 Lambda's en Streams

Java How to Program, 10/e

© Copyright 1992–2015 by Pearson Education, Inc.  
All Rights Reserved.

## Doelstellingen

- Leren wat functioneel programmeren inhoudt en hoe het objectgeoriënteerd programmeren aanvult.
- Functioneel programmeren gebruiken om bepaalde programmeertaken te vereenvoudigen.
- Lambda expressies schrijven die functionele interfaces implementeren.
- Wat zijn streams? Hoe worden stream pipelines gevormd uit streambronnen, intermediate operaties en terminal operaties.
- Uitvoeren van operaties op IntStreams, zoals forEach, count, min, max, sum, average, reduce, filter en sorted.
- Uitvoeren van operaties op Streams, zoals filter, map, sorted, collect, forEach, findFirst, distinct, mapToDouble en reduce.
- Creëren van streams met int waarden (binnen een bepaald bereik en willekeurige).

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

## 17.1 Inleiding

- Tot Java SE 8 ondersteunde Java drie programmeerparadigma's:
  - procedureel programmeren,
  - object-geörienteerd programmeren en
  - generiek programmeren.  
=>Java SE 8 nu ook functioneel programmeren.
- Project Lambda:  
<http://openjdk.java.net/projects/lambda>

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.2 Functioneel Programmeren Technologieën Overzicht

- Tot nu toe: je specificeert **hoe** een taak moet worden uitgevoerd.

```
int sum=0,values[];  
for(int counter=0 ; counter < values.length ; counter++)  
    sum += values[counter];
```
- **External** iteratie
  - Gebruik van een lus om te itereren over een collectie van elementen.
  - Vereist sequentiële benadering van de elementen.
  - Vereist veranderlijke variabelen (sum en counter).

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.2 Functioneel Programmeren Technologieën Overzicht (verv.)

- Functioneel programmeren
  - Specifeer **wat** je wil in een taak, maar niet hoe.
- **Internal iteratie**
  - Laat de bibliotheek de manier bepalen om over een collectie van elementen te itereren.
  - Internal iteratie is gemakkelijker voor parallelle uitvoering.
- Functioneel programmeren legt de klemtoon op immutability, het niet aanpassen van de aangesproken databron.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.2.1 Functionele Interfaces

- Functionele interfaces, ook gekend als single abstract method (SAM) interfaces. (bevatten één abstracte methode)
- Package `java.util.function`
  - 6 basis functional interfaces
  - Figuur 17.2
- Meerdere gespecialiseerde versies van de basis functionele interfaces
  - Te gebruiken met `int`, `long` en `double` primitieve waarden.
- Ook generieke aanpassingen van `Consumer`, `Function` and `Predicate`
  - Voor binaire operaties; methodes met twee argumenten.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

Interface	Description
<code>BinaryOperator&lt;T&gt;</code>	Contains method <code>apply</code> that takes two <code>T</code> arguments, performs an operation on them (such as a calculation) and returns a value of type <code>T</code> . You'll see several examples of <code>BinaryOperators</code> starting in Section 17.3.
<code>Consumer&lt;T&gt;</code>	Contains method <code>accept</code> that takes a <code>T</code> argument and returns <code>void</code> . Performs a task with its <code>T</code> argument, such as outputting the object, invoking a method of the object, etc. You'll see several examples of <code>Consumers</code> starting in Section 17.3.
<code>Function&lt;T, R&gt;</code>	Contains method <code>apply</code> that takes a <code>T</code> argument and returns a value of type <code>R</code> . Calls a method on the <code>T</code> argument and returns that method's result. You'll see several examples of <code>Functions</code> starting in Section 17.5.
<code>Predicate&lt;T&gt;</code>	Contains method <code>test</code> that takes a <code>T</code> argument and returns a <code>boolean</code> . Tests whether the <code>T</code> argument satisfies a condition. You'll see several examples of <code>Predicates</code> starting in Section 17.3.

**Fig. 17.2** | The six basic generic functional interfaces in package `java.util.function`.

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

**HoGent**

Interface	Description
<code>Supplier&lt;T&gt;</code>	Contains method <code>T apply(T t)</code> and produces a value of type <code>T</code> . Often used to create a collection object in which a stream operation's results are placed. You'll see several examples of <code>Suppliers</code> starting in Section 17.7.
<code>UnaryOperator&lt;T&gt;</code>	Contains method <code>get</code> that takes no arguments and returns a value of type <code>T</code> . You'll see several examples of <code>UnaryOperators</code> starting in Section 17.3.

**Fig. 17.2** | The six basic generic functional interfaces in package `java.util.function`.

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

**HoGent**

## 17.2.2 Lambda Expressies

- Lambda expressie
  - anonieme methode
  - snelschrift notatie voor het implementeren van een functionele interface.
- Het type lambda expressie is het type van de functionele interface die de lambda implementeert.
- Kan gebruikt worden waar functionele interfaces worden verwacht.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.2.2 Lambda Expressies (vervolg)

- Een lambda expressie bestaat uit een parameterlijst gevolgd door een pijltoken en een body:  
 $(parameterList) \rightarrow \{statements\}$
- Vb: lambda ontvangt twee ints en geeft hun som terug:  
 $(int x, int y) \rightarrow \{return x + y;\}$

Deze lambda's body is een blok dat één of meerdere statements kan bevatten tussen accolades.

Er zijn meerdere variaties mogelijk:

$(x, y) \rightarrow \{return x + y;\}$   
 $(x, y) \rightarrow x + y$

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.2.2 Lambda Expressies (vervolg)

- Bestaat de parameterlijst uit één parameter, dan mogen de haakjes weg:  
`value -> System.out.printf("%d ", value)`
- Een lambda met lege parameterlijst:  
`() -> System.out.println("Welcome to Lambdas!")`
- Zie verder 17.5.1.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.2.3 Streams

- Streams zijn objecten van
  - klassen die de interface `Stream` (from the package `java.util.stream`) implementeren
  - Eén van de gespecialiseerde stream interfaces voor verwerking van `int`, `long` of `double` waarden
- Stream pipelines
  - Laat elementen een reeks van verwerkingsstappen doorlopen.
  - Pipeline
    - begint met een databron,
    - voert meerdere **intermediate** operaties uit op de elementen van de databron en
    - eindigt met een **terminal** operatie.
  - Wordt gevormd door geketende methode aanroepen.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.2.3 Streams (vervolg)

- Streams bewaren geen data
  - Eenmaal een stream is uitgevoerd kan het niet worden herbruikt, omdat het geen kopij bijhoudt van de originele databron.
- Intermediate (=tussentijdse) operatie
  - specificeert een taak op elementen van een stream en resulteert altijd in een nieuwe stream.
  - Zijn lazy: worden pas uitgevoerd als een terminal operatie wordt aangeroepen.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.2.3 Streams (vervolg)

- Terminal (=eind) operatie
  - Start de verwerking van de stream pipeline's intermediate operaties
  - Creëert een resultaat
  - Zijn eager: voeren de gevraagde operatie uit wanneer ze worden aangeroepen.
- Figuur 17.3 intermediate operaties.
- Figuur 17.4 terminal operaties.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

**Intermediate Stream operations**

<code>filter</code>	Results in a stream containing only the elements that satisfy a condition.
<code>distinct</code>	Results in a stream containing only the unique elements.
<code>limit</code>	Results in a stream with the specified number of elements from the beginning of the original stream.
<code>map</code>	Results in a stream in which each element of the original stream is mapped to a new value (possibly of a different type)—e.g., mapping numeric values to the squares of the numeric values. The new stream has the same number of elements as the original stream.
<code>sorted</code>	Results in a stream in which the elements are in sorted order. The new stream has the same number of elements as the original stream.

**Fig. 17.3** | Common intermediate Stream operations.

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

**HoGent**

**Terminal Stream operations**

<code>foreach</code>	Performs processing on every element in a stream (e.g., display each element).
<b>Reduction operations</b> — <i>Take all values in the stream and return a single value</i>	
<code>average</code>	Calculates the <i>average</i> of the elements in a numeric stream.
<code>count</code>	Returns the <i>number of elements</i> in the stream.
<code>max</code>	Locates the <i>largest</i> value in a numeric stream.
<code>min</code>	Locates the <i>smallest</i> value in a numeric stream.
<code>reduce</code>	Reduces the elements of a collection to a <i>single value</i> using an associative accumulation function (e.g., a lambda that adds two elements).
<b>Mutable reduction operations</b> — <i>Create a container (such as a collection or StringBuilder)</i>	
<code>collect</code>	Creates a <i>new collection</i> of elements containing the results of the stream's prior operations.
<code>toArray</code>	Creates an <i>array</i> containing the results of the stream's prior operations.

**Fig. 17.4** | Common terminal Stream operations.

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

**HoGent**

**Terminal Stream operations**

*Search operations*

<code>findFirst</code>	Finds the <i>first</i> stream element based on the prior intermediate operations; immediately terminates processing of the stream pipeline once such an element is found.
<code>findAny</code>	Finds <i>any</i> stream element based on the prior intermediate operations; immediately terminates processing of the stream pipeline once such an element is found.
<code>anyMatch</code>	Determines whether <i>any</i> stream elements match a specified condition; immediately terminates processing of the stream pipeline if an element matches.
<code>allMatch</code>	Determines whether <i>all</i> of the elements in the stream match a specified condition.

**Fig. 17.4 |** Common terminal Stream operations.

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

HoGent

## 17.3 IntStream Operaties

- Figure 17.5 : bewerkingen op een **IntStream** (package **java.util.stream**) – een gespecialiseerde stream voor het manipuleren van `int` waarden.
- De gebruikte technieken gelden ook voor **LongStreams** and **DoubleStreams** voor respectievelijk `long` en `double` waarden.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```

1 // Fig. 17.5: IntStreamOperations.java
2 // Demonstrating IntStream operations.
3 import java.util.Arrays;
4 import java.util.stream.IntStream;
5
6 public class IntStreamOperations
7 {
8     public static void main(String[] args)
9     {
10         int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};
11
12         // display original values
13         System.out.print("Original values: ");
14         IntStream.of(values)
15             .forEach(value -> System.out.printf("%d ", value));
16         System.out.println();
17

```

**Fig. 17.5** | Demonstrating IntStream operations. (Part 1 of 5.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

```

18     // count, min, max, sum and average of the values
19     System.out.printf("%nCount: %d%n", IntStream.of(values).count());
20     System.out.printf("Min: %d%n",
21         IntStream.of(values).min().getAsInt());
22     System.out.printf("Max: %d%n",
23         IntStream.of(values).max().getAsInt());
24     System.out.printf("Sum: %d%n", IntStream.of(values).sum());
25     System.out.printf("Average: %.2f%n",
26         IntStream.of(values).average().getAsDouble());
27
28     // sum of values with reduce method
29     System.out.printf("%nSum via reduce method: %d%n",
30         IntStream.of(values)
31             .reduce(0, (x, y) -> x + y));
32
33     // sum of squares of values with reduce method
34     System.out.printf("%nSum of squares via reduce method: %d%n",
35         IntStream.of(values)
36             .reduce(0, (x, y) -> x + y * y));

```

**Fig. 17.5** | Demonstrating IntStream operations. (Part 2 of 5.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

```

37      // product of values with reduce method
38      System.out.printf("Product via reduce method: %d%n",
39          IntStream.of(values)
40              .reduce(1, (x, y) -> x * y));
41
42      // even values displayed in sorted order
43      System.out.printf("%nEven values displayed in sorted order: ");
44      IntStream.of(values)
45          .filter(value -> value % 2 == 0)
46          .sorted()
47          .forEach(value -> System.out.printf("%d ", value));
48      System.out.println();
49
50      // odd values multiplied by 10 and displayed in sorted order
51      System.out.printf(
52          "Odd values multiplied by 10 displayed in sorted order: ");
53      IntStream.of(values)
54          .filter(value -> value % 2 != 0)
55          .map(value -> value * 10)
56          .sorted()
57          .forEach(value -> System.out.printf("%d ", value));
58      System.out.println();
59
60

```

**Fig. 17.5** | Demonstrating IntStream operations. (Part 3 of 5.)

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

**HoGent**

```

61      // sum range of integers from 1 to 10, exclusive
62      System.out.printf("%nSum of integers from 1 to 9: %d%n",
63          IntStream.range(1, 10).sum());
64
65      // sum range of integers from 1 to 10, inclusive
66      System.out.printf("Sum of integers from 1 to 10: %d%n",
67          IntStream.rangeClosed(1, 10).sum());
68  }
69 } // end class IntStreamOperations

```

**Fig. 17.5** | Demonstrating IntStream operations. (Part 4 of 5.)

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

**HoGent**

```

Original values: 3 10 6 1 4 8 2 5 9 7
Count: 10
Min: 1
Max: 10
Sum: 55
Average: 5.50

Sum via reduce method: 55
Sum of squares via reduce method: 385
Product via reduce method: 3628800

Even values displayed in sorted order: 2 4 6 8 10
Odd values multiplied by 10 displayed in sorted order: 10 30 50 70 90

Sum of integers from 1 to 9: 45
Sum of integers from 1 to 10: 55

```

**Fig. 17.5** | Demonstrating IntStream operations. (Part 5 of 5.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

### 17.3.1 Creatie IntStream en tonen van zijn waarden met forEach Terminal Operatie

- IntStream static methode of krijgt een int array als argument en geeft een IntStream terug om de waarden in de array te verwerken.
- IntStream methode forEach (terminal operatie) krijgt als argument een object dat de IntConsumer functional interface (package java.util.function) implementeert. Deze interface accept methode krijgt één int waarde en voert er een taak mee uit.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.3.1 Creatie IntStream en tonen van zijn waarden met forEach Terminal Operatie (verv.)

- Compiler kan de types van de lambda's parameters en het type dat teruggeven wordt door de lambda, afleiden uit de context waarin de lambda wordt gebruikt.
- Lambda's kunnen **final** lokale variabelen of effectieve **final** lokale variabelen gebruiken.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.3.1 Creatie IntStream en tonen van zijn waarden met forEach Terminal Operatie (verv.)

- Een lambda gebruikt **this** om te refereren naar het object van de outerklasse.
- De namen voor parameters en variabelen in lambda's mogen niet dezelfde zijn als die van andere lokale variabelen in de lambda's lexical scope.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.3.2 Terminal Operaties count, min, max, sum en average

- Klasse `IntStream` voorziet:
  - `count` geeft aantal elementen terug
  - `min` geeft de kleinste `int` terug
  - `max` geeft de grootste `int` terug
  - `sum` geeft de som van alle `ints` terug
  - `average` geeft een `OptionalDouble` (package `java.util`) terug, die bevat het gemiddelde van de `ints` als een waarde van het type `double`
- Klasse `OptionalDouble`'s `getAsDouble` methode geeft de `double` in het object terug of gooit een `NoSuchElementException`.
  - Om deze exception te voorkomen, kan je de methode `orElse` gebruiken. Deze geeft de `OptionalDouble`'s waarde terug als er een is, of de waarde die je doorgeeft aan `orElse`.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.3.2 Terminal Operaties count, min, max, sum en average (verv.)

- `IntStream` methode `summaryStatistics` voert de `count`, `min`, `max`, `sum` en `average` operaties uit in 1 doorloop en geeft de resultaten terug als een `IntSummaryStatistics` object (package `java.util`).

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.3.3 Terminal Operation reduce

- Je kan je eigen verkortingen definiëren voor een `IntStream` door zijn `reduce` methode aan te roepen.
  - Eerste argument is een waarde die gebruikt wordt als begin van de reduction operatie
  - Tweede argument is een object dat de `IntBinaryOperator` functional interface implementeert.
- Het eerste argument van de methode `reduce` wordt een identity waarde genoemd. Als deze waarde gecombineerd wordt met een stream element, gebruikmakend van `IntBinaryOperator`, dan levert dat de originele waarde van dat element op.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.3.4 Intermediate Operaties: Filteren en sorteren van `IntStream` waarden

- Filteren van elementen volgens een bepaalde voorwaarde.
- `IntStream` methode `filter` ontvangt een object dat de `IntPredicate` functional interface (package `java.util.function`) implementeert.
- `IntStream` methode `sorted` ordert de elementen van de stream in oplopende volgorde(by default).

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.3.4 Intermediate Operaties: Filteren en sorteren van IntStream waarden(verv.)

- Interface `IntPredicate`'s `default` method and voert een logische AND operatie uit met short-circuit evaluatie.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.3.4 Intermediate Operaties: Filteren en sorteren van IntStream waarden(verv.)

- Interface `IntPredicate`'s `default` methode `negate` keert de boolean waarde om.
- Interface `IntPredicate` `default` methode `or` voert een logische OR operatie met short-circuit evaluatie uit.
- Gebruik de interface `IntPredicate` `default` methoden om complexere voorwaarden samen te stellen.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.3.5 Intermediate Operation: Mapping

- Mapping is een intermediate operatie die de elementen van een stream omzet naar nieuwe waarden en een stream met de resultaten creeert.
- `IntStream` methode `map` ontvangt een object dat de `IntUnaryOperator` functional interface (package `java.util.function`) implementeert.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.3.6 Creatie van streams van `ints` met `IntStream` methoden `range` en `rangeClosed`

- `IntStream` methoden `range` en `rangeClosed` produceren elk een geordende reeks van `int` waarden.
  - Beide methoden hebben twee `int` argumenten die het bereik van de waarden voorstellen.
  - Methode `range` produceert een reeks van waarden vanaf het eerste argument tot, niet inbegrepen, het tweede argument.
  - Methode `rangeClosed` produceert een reeks van waarden, beide argumenten inbegrepen.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.4 Stream<Integer> bewerkingen

- Klasse Array's stream methode wordt gebruikt om een Stream te creëren vertrekkend van een array van objecten.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```

1 // Fig. 17.6: ArraysAndStreams.java
2 // Demonstrating lambdas and streams with an array of Integers.
3 import java.util.Arrays;
4 import java.util.Comparator;
5 import java.util.List;
6 import java.util.stream.Collectors;
7
8 public class ArraysAndStreams
9 {
10     public static void main(String[] args)
11     {
12         Integer[] values = {2, 9, 5, 0, 3, 7, 1, 4, 8, 6};
13
14         // display original values
15         System.out.printf("Original values: %s%n", Arrays.asList(values));
16
17         // sort values in ascending order with streams
18         System.out.printf("Sorted values: %s%n",
19             Arrays.stream(values)
20                 .sorted()
21                 .collect(Collectors.toList()));
22

```

**Fig. 17.6** | Demonstrating lambdas and streams with an array of Integers. (Part I of 3.)

```

23 // values greater than 4
24 List<Integer> greaterThan4 =
25     Arrays.stream(values)
26         .filter(value -> value > 4)
27         .collect(Collectors.toList());
28 System.out.printf("Values greater than 4: %s%n", greaterThan4);
29
30 // filter values greater than 4 then sort the results
31 System.out.printf("Sorted values greater than 4: %s%n",
32     Arrays.stream(values)
33         .filter(value -> value > 4)
34         .sorted()
35         .collect(Collectors.toList()));
36
37 // greaterThan4 List sorted with streams
38 System.out.printf(
39     "Values greater than 4 (ascending with streams): %s%n",
40     greaterThan4.stream()
41         .sorted()
42         .collect(Collectors.toList()));
43 }
44 } // end class ArraysAndStreams

```

**Fig. 17.6** | Demonstrating lambdas and streams with an array of `Integers`. (Part 2 of 3.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

```

Original values: [2, 9, 5, 0, 3, 7, 1, 4, 8, 6]
Sorted values: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Values greater than 4: [9, 5, 7, 8, 6]
Sorted values greater than 4: [5, 6, 7, 8, 9]
Values greater than 4 (ascending with streams): [5, 6, 7, 8, 9]

```

**Fig. 17.6** | Demonstrating lambdas and streams with an array of `Integers`. (Part 3 of 3.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

### 17.4.1 Creatie van een Stream<Integer>

- Interface Stream (pakket java.util.stream) is een generieke interface voor het uitvoeren van stream operaties op objecten. De typen van de objecten die verwerkt worden, worden bepaald door de Stream's bron.
- Klasse Arrays biedt overloaded stream methoden voor het creëren van IntStreams, LongStreams en DoubleStreams uit int, long en double arrays of reeksen van elementen uit de arrays.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.4.2 Sorteren van een Stream en verzamelen van de resultaten.

- Stream methode sorted sorteert de elementen van een stream in oplopende volgorde by default.
- Voor het creëren van een verzameling die de resultaten van een stream pipeline bevat, kan je de Stream methode collect (terminal operatie) gebruiken.
- Methode collect met één argument krijgt een object dat de interface Collector (package java.util.stream) implementeert, die specificeert hoe de veranderlijke reductie moet worden uitgevoerd.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.4.2 Sorteren van een Stream en verzamelen van de resultaten (verv.)

- Klasse `Collectors` (package `java.util.stream`) voorziet `static` methoden die voorgedefinieerde `Collector` implementaties teruggegeven.
- `Collectors` methode `toList` zet een `Stream<T>` om in een `List<T>` collectie.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.4.3 Filteren van een Stream en de resultaten bewaren

- `Stream` methode `filter` krijgt een `Predicate` en resulteert in een stream van objecten die aan de `Predicate` voldoen.
- `Predicate` methode `test` geeft een boolean terug, die weergeeft of het argument voldoet aan de conditie. Interface `Predicate` heeft ook de methoden `and`, `negate` en `or`.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

#### 17.4.4 Sorteren van vooraf verzamelde resultaten

- Zodra u de resultaten van een Stream pipeline in een verzameling hebt geplaatst, kunt u een nieuwe Stream vanuit de collectie maken voor het uitvoeren van extra Stream operaties op de eerdere resultaten.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

#### 17.5 Stream<String> bewerkingen

- Figuur 17.7 Stream<String>.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```

1 // Fig. 17.7: ArraysAndStreams2.java
2 // Demonstrating lambdas and streams with an array of Strings.
3 import java.util.Arrays;
4 import java.util.Comparator;
5 import java.util.stream.Collectors;
6
7 public class ArraysAndStreams2
8 {
9     public static void main(String[] args)
10    {
11        String[] strings =
12            {"Red", "orange", "Yellow", "green", "Blue", "indigo", "Violet"};
13
14        // display original strings
15        System.out.printf("Original strings: %s%n", Arrays.asList(strings));
16
17        // strings in uppercase
18        System.out.printf("strings in uppercase: %s%n",
19            Arrays.stream(strings)
20                .map(String::toUpperCase)
21                .collect(Collectors.toList()));
22

```

**Fig. 17.7** | Demonstrating lambdas and streams with an array of Strings. (Part I of 2.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

```

23     // strings less than "n" (case insensitive) sorted ascending
24     System.out.printf("strings greater than m sorted ascending: %s%n",
25         Arrays.stream(strings)
26             .filter(s -> s.compareToIgnoreCase("n") < 0)
27             .sorted(String.CASE_INSENSITIVE_ORDER)
28             .collect(Collectors.toList()));
29
30     // strings less than "n" (case insensitive) sorted descending
31     System.out.printf("strings greater than m sorted descending: %s%n",
32         Arrays.stream(strings)
33             .filter(s -> s.compareToIgnoreCase("n") < 0)
34             .sorted(String.CASE_INSENSITIVE_ORDER.reversed())
35             .collect(Collectors.toList()));
36     }
37 } // end class ArraysAndStreams2

```

```

Original strings: [Red, orange, Yellow, green, Blue, indigo, Violet]
strings in uppercase: [RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET]
strings greater than m sorted ascending: [orange, Red, Violet, Yellow]
strings greater than m sorted descending: [Yellow, Violet, Red, orange]

```

**Fig. 17.7** | Demonstrating lambdas and streams with an array of Strings. (Part 2 of 2.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

### 17.5.1 Omzetten van Strings naar hoofdletters gebruikmakend van een methode referentie

- Stream methode map zet ieder element om naar een nieuwe waarde en maakt een nieuwe stream met hetzelfde aantal elementen als de originele stream.
- Een methodereferentie is een snelschrift notatie voor een lambda expressie.
- *ClassName::instanceMethodName* stelt een methodereferentie voor van een instantiemethode van de klasse.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.5.1 Omzetten van Strings naar hoofdletters gebruikmakend van een methode referentie (verv.)

- *objectName::instanceMethodName* stelt een methodereferentie voor, voor een instance methode die aangeroepen wordt op een specifiek object.
- *ClassName::staticMethodName* stelt een methodereferentie voor, voor een static methode van een klasse.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.5.1 Omzetten van Strings naar hoofdletters gebruikmakend van een methode referentie (verv.)

- *ClassName*::*new* stelt een constructor referentie voor.
- Figuur 17.8 toont de vier methode referentie- types.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

Lambda	Description
<code>String::toUpperCase</code>	Method reference for an instance method of a class. Creates a one-parameter lambda that invokes the instance method on the lambda's argument and returns the method's result. Used in Fig. 17.7.
<code>System.out::println</code>	Method reference for an instance method that should be called on a specific object. Creates a one-parameter lambda that invokes the instance method on the specified object—passing the lambda's argument to the instance method—and returns the method's result. Used in Fig. 17.10.
<code>Math::sqrt</code>	Method reference for a static method of a class. Creates a one-parameter lambda in which the lambda's argument is passed to the specified a static method and the lambda returns the method's result.
<code>TreeMap::new</code>	Constructor reference. Creates a lambda that invokes the no-argument constructor of the specified class to create and initialize a new object of that class. Used in Fig. 17.17.

**Fig. 17.8** | Types of method references.

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

### 17.5.2 Filteren van Strings en daarna sorteren in Case-Insensitive oplopende volgorde

- Stream methode sorted kan een Comparator als argument ontvangen, om zo de sorteervolgorde vast te leggen.
- By default, methode sorted gebruikt de natuurlijke volgorde voor de stream's element type.
- Voor Strings, de natuurlijke volgorde is case sensitive, dit betekent dat "Z" is kleiner dan "a".

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

### 17.5.2 Filteren van Strings en daarna sorteren in Case-Insensitive aflopende volgorde

- Functional interface Comparator's default methode reversed keert een bestaande Comparator's volgorde om.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.6 Stream<Employee> Manipulations

- Het voorbeeld in Figs. 17.9–17.16 toont verscheidene lambda en stream mogelijkheden die een Stream<Employee> gebruiken.
- Klasse Employee (Fig. 17.9) stelt een werknemer voor met een voornaam, familienaam, een salaris en zijn afdeling.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```

1 // Fig. 17.9: Employee.java
2 // Employee class.
3 public class Employee
4 {
5     private String firstName;
6     private String lastName;
7     private double salary;
8     private String department;
9
10    // constructor
11    public Employee(String firstName, String lastName,
12                    double salary, String department)
13    {
14        this.firstName = firstName;
15        this.lastName = lastName;
16        this.salary = salary;
17        this.department = department;
18    }
19
20    // set firstName
21    public void setFirstName(String firstName)
22    {
23        this.firstName = firstName;
24    }

```

**Fig. 17.9** | Employee class for use in Figs. 17.10–17.16. (Part I of 4.)

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

HoGent

```

25
26     // get firstName
27     public String getFirstName()
28     {
29         return firstName;
30     }
31
32     // set lastName
33     public void setLastName(String lastName)
34     {
35         this.lastName = lastName;
36     }
37
38     // get lastName
39     public String getLastname()
40     {
41         return lastName;
42     }
43
44     // set salary
45     public void setSalary(double salary)
46     {
47         this.salary = salary;
48     }

```

**Fig. 17.9** | Employee class for use in Figs. 17.10–17.16. (Part 2 of 4.)

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

**HoGent**

```

49
50     // get salary
51     public double getSalary()
52     {
53         return salary;
54     }
55
56     // set department
57     public void setDepartment(String department)
58     {
59         this.department = department;
60     }
61
62     // get department
63     public String getDepartment()
64     {
65         return department;
66     }
67
68     // return Employee's first and last name combined
69     public String getName()
70     {
71         return String.format("%s %s", getFirstName(), getLastname());
72     }

```

**Fig. 17.9** | Employee class for use in Figs. 17.10–17.16. (Part 3 of 4.)

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

**HoGent**

```

73
74     // return a String containing the Employee's information
75     @Override
76     public String toString()
77     {
78         return String.format("%-8s %-8s %8.2f %s",
79             getFirstName(), getLastName(), getSalary(), getDepartment());
80     } // end method toString
81 } // end class Employee

```

**Fig. 17.9** | Employee class for use in Figs. 17.10–17.16. (Part 4 of 4.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

## 17.6.1 Creatie en tonen van een List<Employee>

- Instantie methode referentie `System.out::println` wordt doorgegeven naar de Stream methode `forEach`.  
Deze wordt door de compiler omgezet naar een object dat de `Consumer` functional interface implementeert.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.6.1 Creatie en tonen van een List<Employee> (verv.)

- Figuur 17.10 maakt een array van Employees en haalt de List view.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```

1 // Fig. 17.10: ProcessingEmployees.java
2 // Processing streams of Employee objects.
3 import java.util.Arrays;
4 import java.util.Comparator;
5 import java.util.List;
6 import java.util.Map;
7 import java.util.TreeMap;
8 import java.util.function.Function;
9 import java.util.function.Predicate;
10 import java.util.stream.Collectors;
11
12 public class ProcessingEmployees
13 {
14     public static void main(String[] args)
15     {
16         // initialize array of Employees
17         Employee[] employees = {
18             new Employee("Jason", "Red", 5000, "IT"),
19             new Employee("Ashley", "Green", 7600, "IT"),
20             new Employee("Matthew", "Indigo", 3587.5, "Sales"),
21             new Employee("James", "Indigo", 4700.77, "Marketing"),
22             new Employee("Luke", "Indigo", 6200, "IT"),
23             new Employee("Jason", "Blue", 3200, "Sales"),
24             new Employee("Wendy", "Brown", 4236.4, "Marketing")};

```

**Fig. 17.10** | Creating an array of Employees, converting it to a List and displaying the List (Part 1 of 2)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

```

25
26     // get List view of the Employees
27     List<Employee> list = Arrays.asList(employees);
28
29     // display all Employees
30     System.out.println("Complete Employee list:");
31     list.stream().forEach(System.out::println);
32

```

```

Complete Employee list:
Jason    Red      5000.00   IT
Ashley   Green    7600.00   IT
Matthew  Indigo   3587.50   Sales
James    Indigo   4700.77   Marketing
Luke    Indigo    6200.00   IT
Jason    Blue     3200.00   Sales
Wendy   Brown    4236.40   Marketing

```

**Fig. 17.10** | Creating an array of Employees, converting it to a List and displaying the List. (Part 2 of 2.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

## 17.6.2 Filteren van Employees met salarissen in bepaalde bereiken

- Figuur 17.11 toont het filteren van Employees met een object dat de functional interface `Predicate<Employee>` implementeert, die gedefinieerd wordt met een lambda.
- Om een lambda te hergebruiken, kan je het toekennen aan een variabele van het juiste functional interface type.
- De `Comparator` interface's `static` methode `comparing` ontvangt een `Function` dat gebruikt wordt om een waarde uit een object in de stream te halen, dat verder gebruikt wordt in vergelijkingen en een `Comparator` object teruggeeft.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```

33 // Predicate that returns true for salaries in the range $4000-$6000
34 Predicate<Employee> fourToSixThousand =
35     e -> (e.getSalary() >= 4000 && e.getSalary() <= 6000);
36
37 // Display Employees with salaries in the range $4000-$6000
38 // sorted into ascending order by salary
39 System.out.printf(
40     "%nEmployees earning $4000-$6000 per month sorted by salary:%n");
41 list.stream()
42     .filter(fourToSixThousand)
43     .sorted(Comparator.comparing(Employee::getSalary))
44     .forEach(System.out::println);
45
46 // Display first Employee with salary in the range $4000-$6000
47 System.out.printf("%nFirst employee who earns $4000-$6000:%n%s%n",
48     list.stream()
49     .filter(fourToSixThousand)
50     .findFirst()
51     .get());
52

```

**Fig. 17.11** | Filtering Employees with salaries in the range \$4000–\$6000. (Part 1 of 2.)

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

**HoGent**

Employees earning \$4000–\$6000 per month sorted by salary:  
Wendy Brown 4236.40 Marketing  
James Indigo 4700.77 Marketing  
Jason Red 5000.00 IT

First employee who earns \$4000–\$6000:  
Jason Red 5000.00 IT

**Fig. 17.11** | Filtering Employees with salaries in the range \$4000–\$6000. (Part 2 of 2.)

**HoGent**

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

## 17.6.2 Filteren van Employees met salarissen in bepaalde bereiken (verv.)

- Stream methode `findFirst` is een short-circuit terminal operation die de stream pipeline uitvoert en stopt met de verwerking vanaf het moment dat het eerste object in de stream pipeline is gevonden.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.6.3 Sorteren van Employees op basis van meerdere velden

- Figuur 17.12 toont hoe streams te gebruiken om objecten te sorteren op meerdere velden.
- Om objecten te sorteren op twee velden, creëer je een `Comparator` die twee `Functions` gebruikt.
- Eerst roep je de `Comparator` methode `comparing` op om een `Comparator` met een eerste `Function` te creëren.
- Op de resulterende `Comparator`, roep je de methode `thenComparing` met de tweede `Function`.
- De resulterende `Comparator` vergelijkt de objecten volgens de eerste `Function` en, voor objecten die gelijk zijn, vervolgens op de tweede `Function`.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```

53  // Functions for getting first and last names from an Employee
54  Function<Employee, String> byFirstName = Employee::getFirstName;
55  Function<Employee, String> byLastName = Employee::getLastName;
56
57  // Comparator for comparing Employees by first name then last name
58  Comparator<Employee> lastThenFirst =
59      Comparator.comparing(byLastName).thenComparing(byFirstName);
60
61  // sort employees by last name, then first name
62  System.out.printf(
63      "%nEmployees in ascending order by last name then first:%n");
64  list.stream()
65      .sorted(lastThenFirst)
66      .forEach(System.out::println);
67
68  // sort employees in descending order by last name, then first name
69  System.out.printf(
70      "%nEmployees in descending order by last name then first:%n");
71  list.stream()
72      .sorted(lastThenFirst.reversed())
73      .forEach(System.out::println);
74

```

**Fig. 17.12** | Sorting Employees by last name then first name. (Part I of 2.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

Employees in ascending order by last name then first:
Jason    Blue    3200.00    Sales
Wendy    Brown    4236.40    Marketing
Ashley    Green    7600.00    IT
James    Indigo    4700.77    Marketing
Luke    Indigo    6200.00    IT
Matthew    Indigo    3587.50    Sales
Jason    Red    5000.00    IT

Employees in descending order by last name then first:
Jason    Red    5000.00    IT
Matthew    Indigo    3587.50    Sales
Luke    Indigo    6200.00    IT
James    Indigo    4700.77    Marketing
Ashley    Green    7600.00    IT
Wendy    Brown    4236.40    Marketing
Jason    Blue    3200.00    Sales

**Fig. 17.12** | Sorting Employees by last name then first name. (Part 2 of 2.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

## 17.6.4 Afbeelden Employees naar unieke familienaam Strings

- Figuur 17.13 illustreert hoe objecten van het ene type (`Employee`) kunnen worden afgebeeld op objecten van een ander type (`String`).
- Je kan objecten van een stream afbeelden naar een andere stream van verschillend type, met hetzelfde aantal elementen als in de originele.
- Stream methode `distinct` verwijdert dubbele objecten uit de stream.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```

75      // display unique employee last names sorted
76      System.out.printf("%nUnique employee last names:%n");
77      list.stream()
78          .map(Employee::getLastName)
79          .distinct()
80          .sorted()
81          .forEach(System.out::println);
82
83      // display only first and last names
84      System.out.printf(
85          "%nEmployee names in order by last name then first name:%n");
86      list.stream()
87          .sorted(lastThenFirst)
88          .map(Employee::getName)
89          .forEach(System.out::println);
90

```

**Fig. 17.13** | Mapping Employee objects to last names and whole names. (Part I of 2.)

```

Unique employee last names:
Blue
Brown
Green
Indigo
Red

Employee names in order by last name then first name:
Jason Blue
Wendy Brown
Ashley Green
James Indigo
Luke Indigo
Matthew Indigo
Jason Red

```

**Fig. 17.13** | Mapping Employee objects to last names and whole names. (Part 2 of 2.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

## 17.6.5 Groeperen van Employees per departement

- Figuur 17.14 gebruikt de Stream methode collect om Employees per departement te groeperen.
- Collectors static methode groupingBy met één argument krijgt een Function binnen die de objecten in de stream klassificeert – de teruggegeven waarden door deze functie worden als sleutel gebruikt in een Map.
- Map methode forEach voert een operatie uit op ieder sleutel-waarde paar.
  - Ontvangt een object dat de functional interface BiConsumer implementeert.
  - BiConsumer's accept methode heeft twee parameters.
  - Voor Maps, de eerste stelt de sleutel voor en de tweede de corresponderende waarde.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```

91     // group Employees by department
92     System.out.printf("%nEmployees by department:%n");
93     Map<String, List<Employee>> groupedByDepartment =
94         list.stream()
95             .collect(Collectors.groupingBy(Employee::getDepartment));
96     groupedByDepartment.forEach(
97         (department, employeesInDepartment) ->
98         {
99             System.out.println(department);
100            employeesInDepartment.forEach(
101                employee -> System.out.printf("    %s%n", employee));
102        }
103    );
104

```

**Fig. 17.14** | Grouping Employees by department. (Part I of 2.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

Employees by department:				
Sales				
Matthew	Indigo	3587.50	Sales	
Jason	Blue	3200.00	Sales	
IT				
Jason	Red	5000.00	IT	
Ashley	Green	7600.00	IT	
Luke	Indigo	6200.00	IT	
Marketing				
James	Indigo	4700.77	Marketing	
Wendy	Brown	4236.40	Marketing	

**Fig. 17.14** | Grouping Employees by department. (Part 2 of 2.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

## 17.6.6 Tellen van Employees in elk departement

- Figuur 17.15 gebruikt ook `Stream` methode `collect` en `Collectors static` methode `groupingBy`, maar nu om het aantal `Employees` in elk departement te tellen.
- `Collectors static` methode `groupingBy` met twee argumenten ontvangt een `Function` dat de objecten in de stream bepaalt en een andere `Collector` (de downstream `Collector`).
- `Collectors static` methode `counting` geeft een `Collector` terug die het aantal objecten die voldoen aan de voorwaarde teruggeeft.

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

```

105    // count number of Employees in each department
106    System.out.printf("%nCount of Employees by department:%n");
107    Map<String, Long> employeeCountByDepartment =
108        list.stream()
109            .collect(Collectors.groupingBy(Employee::getDepartment,
110                Collectors.counting()));
111    employeeCountByDepartment.forEach(
112        (department, count) -> System.out.printf(
113            "%s has %d employee(s)%n", department, count));
114

```

```

Count of Employees by department:
IT has 3 employee(s)
Marketing has 2 employee(s)
Sales has 2 employee(s)

```

**Fig. 17.15** | Counting the number of Employees in each department.

## 17.6.7 Sommeren en gemiddelde van Employee salarissen

- Figuur 17.16 toont de Stream methode **mapToDouble**, die beeldt objecten af op double waarden en geeft een DoubleStream terug.
- Stream methode **mapToDouble** ontvangt een object dat de functionele interface **ToDoubleFunction** (package **java.util.function**) implementeert .
  - Deze interface's **applyAsDouble** methode geeft een double waarde terug.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```

115    // sum of Employee salaries with DoubleStream sum method
116    System.out.printf(
117        "%nSum of Employees' salaries (via sum method): %.2f%n",
118        list.stream()
119            .mapToDouble(Employee::getSalary)
120            .sum());
121
122    // calculate sum of Employee salaries with Stream reduce method
123    System.out.printf(
124        "Sum of Employees' salaries (via reduce method): %.2f%n",
125        list.stream()
126            .mapToDouble(Employee::getSalary)
127            .reduce(0, (value1, value2) -> value1 + value2));
128
129    // average of Employee salaries with DoubleStream average method
130    System.out.printf("Average of Employees' salaries: %.2f%n",
131        list.stream()
132            .mapToDouble(Employee::getSalary)
133            .average()
134            .getAsDouble());
135    } // end main
136 } // end class ProcessingEmployees

```

**Fig. 17.16** | Summing and averaging Employee salaries. (Part I of 2.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

```
Sum of Employees' salaries (via sum method): 34524.67
Sum of Employees' salaries (via reduce method): 34525.67
Average of Employees' salaries: 4932.10
```

**Fig. 17.16** | Summing and averaging Employee salaries. (Part 2 of 2.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

## 17.7 Creatie van Stream<String> vanuit een bestand

- Figuur 17.17 gebruikt lambda's en streams om het aantal voorkomens van ieder woord in een bestand samen te vatten. De samenvatting toont de woorden in alfabetische volgorde gegroepeerd per beginletter.
- Files methode lines creëert een Stream<String> voor het lezen van lijnen tekst uit een bestand.
- Stream methode flatMap ontvangt een Function die een object afbeeldt in een stream, bijv. Een lijn van tekst in woorden.
- Pattern methode splitAsStream gebruikt een reguliere expressie om een String op te delen.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```

1 // Fig. 17.17: StreamOfLines.java
2 // Counting word occurrences in a text file.
3 import java.io.IOException;
4 import java.nio.file.Files;
5 import java.nio.file.Paths;
6 import java.util.Map;
7 import java.util.TreeMap;
8 import java.util.regex.Pattern;
9 import java.util.stream.Collectors;
10
11 public class StreamOfLines
12 {
13     public static void main(String[] args) throws IOException
14     {
15         // Regex that matches one or more consecutive whitespace characters
16         Pattern pattern = Pattern.compile("\\s+");
17
18         // count occurrences of each word in a Stream<String> sorted by word
19         Map<String, Long> wordCounts =
20             Files.lines(Paths.get("Chapter2Paragraph.txt"))
21                 .map(line -> line.replaceAll("(?!')\\p{P}", ""))
22                 .flatMap(line -> pattern.splitAsStream(line))
23                 .collect(Collectors.groupingBy(String::toLowerCase,
24                     TreeMap::new, Collectors.counting()));

```

**Fig. 17.17** | Counting word occurrences in a text file. (Part 1 of 2.)

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

**HoGent**

```

25
26     // display the words grouped by starting letter
27     wordCounts.entrySet()
28         .stream()
29         .collect(
30             Collectors.groupingBy(entry -> entry.getKey().charAt(0),
31                 TreeMap::new, Collectors.toList()))
32         .forEach((letter, wordList) ->
33         {
34             System.out.printf("%n%C%n", letter);
35             wordList.stream().forEach(word -> System.out.printf(
36                 "%13s: %d%n", word.getKey(), word.getValue()));
37         });
38     }
39 } // end class StreamOfLines

```

**Fig. 17.17** | Counting word occurrences in a text file. (Part 2 of 2.)

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

**HoGent**

A a: 2 and: 3 application: 2 arithmetic: 1	I inputs: 1 instruct: 1 introduces: 1	R result: 1 results: 2 run: 1
B begin: 1	J java: 1 jdk: 1	S save: 1 screen: 1 show: 1 sum: 1
C calculates: 1 calculations: 1 chapter: 1 chapters: 1 commandline: 1 compares: 1 comparison: 1 compile: 1 computer: 1	L last: 1 later: 1 learn: 1	T that: 3 the: 7 their: 2 then: 2 this: 2 to: 4 tools: 1
	M make: 1 messages: 2	
	N	

**Fig. 17.18** | Output for the program of Fig. 17.17 arranged in three columns.

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

D decisions: 1 demonstrates: 1 display: 1 displays: 2	O numbers: 2 obtains: 1 of: 1 on: 1 output: 1	U two: 2 use: 2 user: 1
E example: 1 examples: 1	P perform: 1 present: 1 program: 1 programming: 1 programs: 2	W we: 2 with: 1
F for: 1 from: 1		Y you'll: 2
H how: 2		

**Fig. 17.18** | Output for the program of Fig. 17.17 arranged in three columns.

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

## 17.7 Creatie van Stream<String> vanuit een bestand

- Collectors methode groupingBy met drie argumenten ontvangt een classifier, een Map factory en een downstream Collector.
  - De classifier is een Function dat objecten teruggeeft, die als sleutels in de resulterende Map worden gebruikt.
  - De Map factory is een object dat de interface Supplier implementeert en een nieuwe Map collectie teruggeeft.
  - De downstream Collector bepaalt hoe de elementen per groep worden verzameld.
- Map methode entrySet geeft een Set van Map.Entry objecten terug, die bevat de Map's sleutel-waarde koppels.
- Set methode stream geeft een stream terug om de Set's elementen te bewerken.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.8 Genereren van streams met willekeurige waarden

- In Fig. 7.7 wordt Fig. 6.7 opnieuw geprogrammeerd, waarbij het volledige switch statement vervangen wordt door één enkel statement, dat de tellers in een array verhoogt.
- Beide versies gebruiken veranderlijke variabelen om de external iteratie te controleren en de resultaten samen te vatten.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.8 Genereren van streams met willekeurige waarden (verv.)

- Figuur 17.19 herprogrammeert de programma's met een *single statement* die alles uitvoert, gebruikmakend van lambda's, streams, internal iteratie en geen veranderlijke variabelen om de dobbelsteen 6.000.000 keren te gooien, de frequenties te bepalen en de resultaten te tonen.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```

1 // Fig. 17.19: RandomIntStream.java
2 // Rolling a die 6,000,000 times with streams
3 import java.security.SecureRandom;
4 import java.util.Map;
5 import java.util.function.Function;
6 import java.util.stream.IntStream;
7 import java.util.stream.Collectors;
8
9 public class RandomIntStream
10 {
11     public static void main(String[] args)
12     {
13         SecureRandom random = new SecureRandom();
14
15         // roll a die 6,000,000 times and summarize the results
16         System.out.printf("%-6s%s%n", "Face", "Frequency");
17         random.ints(6_000_000, 1, 7)
18             .boxed()
19             .collect(Collectors.groupingBy(Function.identity(),
20                 Collectors.counting()))
21             .forEach((face, frequency) ->
22                 System.out.printf("%-6d%d%n", face, frequency));
23     }
24 } // end class RandomIntStream

```

**Fig. 17.19** | Rolling a die 6,000,000 times with streams. (Part I of 2.)  
 © Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

HoGent

Face	Frequency
1	999339
2	999937
3	1000302
4	999323
5	1000183
6	1000916

**Fig. 17.19** | Rolling a die 6,000,000 times with streams. (Part 2 of 2.)

HoGent

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

## 17.8 Genereren van streams met willekeurige waarden (verv.)

- Klasse `SecureRandom`'s methoden `ints`, `longs` en `doubles` (afgeleid van klasse `Random`) geven respectievelijk `IntStream`, `LongStream` en `DoubleStream`, streams van willekeurige getallen terug.
- Methode `ints` zonder argumenten creëert een `IntStream` voor een oneindige (final) stream van random `int` waarden.
- Een oneindige stream is een stream met een onbekend aantal elementen – er wordt een terminal operatie gebruikt om de verwerking op een infinite stream te vervolledigen.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.8 Genereren van streams met willekeurige waarden (verv.)

- Methode ints met een long argument creëert een IntStream met een specifiek aantal willekeurige int waarden.
- Methode ints met twee int argumenten creëert een IntStream voor een oneindige stream van random int waarden in het bereik [eerste argument, tweede argument[.
- Methode ints met een long en twee int argumenten creëert een IntStream met een specifiek aantal willekeurige int waarden in het bereik [eerste argument, tweede argument[.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.8 Genereren van streams met willekeurige waarden (verv.)

- Omzetten van een IntStream naar een Stream<Integer> => IntStream methode boxed.
- Function static methode identity creëert een Function die enkel het argument teruggeeft.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.9 Lambda Event Handlers

- Sommige event-listener interfaces zijn functionele interfaces. In het geval van dergelijke interfaces, kan je de event handlers implementeren met lambda's.
- Voor een eenvoudige event handler kan een lambda het aantal lijnen code enorm reduceren.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.10 Bijkomende nota's Java SE 8 Interfaces

- Functionele interfaces moeten enkel één `abstract` methode bevatten, maar mogen ook `default`- methoden en `static` methoden bevatten die volledig geïmplementeerd worden in de interface declaraties.
- Als een klasse een interface met `default` methoden ondertekent en hen niet overschrijft, dan erft de klasse de `default` methode implementaties.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

## 17.10 Bijkomende nota's Java SE 8 Interfaces (verv.)

- Als een klasse dezelfde `default` methode van twee interfaces erft, dan moet de klasse die methode overschrijven. Anders genereert de compiler een compilatiefout.
- Je kan je eigen functionele interfaces creëren door ervoor te zorgen dat deze één enkele `abstract`- methode en geen of meerdere `default` or `static` methoden bevatten.
- `@FunctionalInterface` annotatie.  
De compiler controleert of er maar één `abstract`- methode is; anders genereert de compiler een compilatiefout.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.